When Z3 Met Yices

Will Blair and Nikka Ghalili
Formal Methods Technical Report

Abstract

Yices and Z3 share very similar beginnings. Z3 has since evolved to be a flagship solver in the field. We set out to find a complement solver to Z3 in Yices, but benchmarks revealed that it has little to offer. We also looked at Yices 2, a domain specific prototype aimed at redesigning Yices. Yices 2 proved to be faster than Z3 with many formulas, however it lacked much of the functionality of Z3. We used Yices 2 and Z3 to build a composite solver that is a frequently faster version of Z3. However, Yices 2 and Z3 do not speak a common language out of the box, so a SMT-LIB 2 parser was integrated with Yices 2 so that it could accept smt2 commands. A more ideal composite solver is also outlined that would parallelize proofs between Z3 and Yices 2.

1 Overview

In software verification we encounter many different constraints we may want to prove valid. This process transforms abstractions and models representing our programs into logical formulas. If we restrict ourselves to propositional logic, we can employ a SAT solver to check whether our constraints are valid. This is not entirely practical, as realistic systems are difficult to express using solely boolean terms. The SMT class of solvers allows us to check for satisfiability of formulas modulo first-order theories. Using theory combination techniques, we can verify systems using many-sorted first order logic where our terms are not constrained to a single domain.

Though this is exciting, the solvers themselves are low level tools and features vary depending on the intended application. To make matters worse, SMT solvers do not all accept the same input despite the efforts of SMT-LIB to advocate a more unified interface. Integrating SMT solvers into one composite is a method that allows end users to leverage the power of domain specific solvers without needing to understand each one in depth. Whether someone wants to verify a hypervisor or automatically prove the next great theorem, their main concern is deciding formulas correctly and, if at all possible, quickly.

In this project, we evaluated two state of the art SMT solvers Z3, and Yices. In this evaluation, we sought to define the set of formulas each solver is suited for in order to build a composite solver that can solve more formulas than either could on their own. In our evaluation, we discovered that the set of formulas both solvers could handle was difficult to define given their common features. To circumvent this issue, we define an SMT solver's capability in terms of its efficiency.

Using this definition we discovered that a comparison between Yices 1 and Z3 is not that rich; Z3 consistently outperforms Yices in almost every logic. Yices 2, on the other hand, is a less featureful SMT prototype that we found offers some significant boost in efficiency over Z3. Using our results from our capability and performance experiments, we make the case that Yices 2 can act as an "optimizer" for Z3. As additional evidence for this claim, we present a case study of Yices 2 proving linear inequalities from constraints in the Applied Type System programming language.

Unfortunately, the only input both accept is the now deprecated SMT-LIB 1.2 language. To make this composite fit for current use, we integrated Yices 2's C API into an open source SMT-LIB 2.0 parser. We will give motivation for supporting SMT-LIB 2.0, and then provide details on how to run our composite solver.

Finally, we reflect on the shortcomings of our composite sketch a more ideal approach. Though we were unable to implement it, we feel that leveraging the functional semantics of the input language and incremental support allows us to support a continuation style in solving an SMT-LIB file. This could bring greater parallelism to SMT solving in a solver independent way.

Figure 1: Z3 and Yices will match all possible Nats to try to prove this.

```
(declare-datatypes () ((Nat zero (succ (pred Nat)))))
(declare-fun p (Nat) Bool)
(assert (p zero))
(assert (forall ((x Nat)) (implies (p (pred x)) (p x))))
(assert (not (forall ((x Nat)) (p x))))
(check-sat)
```

1.1 History

In 2006, Bruno Dutertre and Leonardo de Moura introduced Yices as an efficient SMT solver that made up the core of Stanford Research Institute's suite of verification systems. The solver featured a novel simplex based algorithm integrated with the classic DPLL procedure that Dutertre and De Moura published that same year[4]. This provided a theory for linear and real arithmetic that established Yices as the top SMT solver.

In 2008, De Moura with Nikolaj Bjørner published the first paper on Microsoft's SMT solver, Z3[2]. Like SRI did with Yices, Microsoft began integrating Z3 into their software stack for static analysis and verification. Among the new things incorporated in Z3 were a novel theory combination engine and quantifier instantion algorithm both developed by De Moura and Bjørner. This was in contrast to Yices' more traditional approach of using the Nelson Oppen algorithm and regular E-graph matching. In the next few years, Z3 began to assert itself as the top SMT solver in the SMT-COMP contests with Yices not participating starting with the introduction of the new SMT-LIB 2 syntax in 2010. Z3 performs so well that even though it did not compete in the 2012 competition, its 2011 scores won most of the logic benchmarks [1]. Naturally, we would like to show places where Yices can improve on the current flagship SMT solver Z3.

We introduce the history here to show that Yices and Z3 share a common lineage. Indeed, several of their important solving techniques (arithmetic solver, unsat solvers) are similar, if not the same. Therefore, we stress the importance of looking at efficiency given that it is unlikely that they decide incredibly different sets of logics.

1.2 Capability

If we define power as the set of logics from which each solver can handle a reasonable number of formulas, we would have a droll comparison. Both Yices 1 and Z3 support the standard set of SMT-LIB logics with exception to those involving non linear arithmetic, which only Z3 can handle. We claim that the practical power of a solver is tied to its efficiency.

In our experience an SMT solver will only answer I dont know if the formula given matches its known unknowns, or it times out. For example, if we give Yices a formula that contains non-linear arithmetic, it will fail immediately during parsing. On the other hand, if I give it a formula that requires proof by structural induction, it will run forever in a matching loop. Figure 1.[6] shows the interaction that will cause this behavior in both Yices and Z3. Nonetheless, it demonstrates that the solver "thinks" it can decide the formula, where as it may just run to the timeout.

The power of a solver must then be related to its timeout value. We define the set of formulas a solver can decide as those it can solve within t seconds. Through running each logic available to SMT solvers on both Z3 and Yices, we hope to arrive at the optimal value of t that maximizes the formulas we can solve within a reasonable amoutn of time time. Of course, what we define as "reasonable" is highly subjective and in the following section we choose a variety of timeouts to show what can be solved and in how long.

This helps us navigate the decision landscape for each solver and adds to our comparison. Given that Yices and Z3 have large popularity and similar implementation, it is not surprising they support overlapping logics. Finding what each may solve within some timeout helps us understand which logics are better suited for which solver. If Z3 can decide most of a benchmark in some time t and the Yices cannot, then we could say that logic belongs to Z3.

2 Experiments

For our benchmarks we chose the SMT-LIB 1.2 suite given its large coverage of the logics our SMT solvers can support. In addition, it is the only suite understood by both solvers on their own which meant we were able to spend more time benchmarking at the beginning and less time trying to add features to solvers we were initially completely unfamilar with.

For both of the capability and performance tests, we ran Z3 and Yices as the executables given by the authors. This minimized the error in our tests given that the programming API for both solvers is non-trivial for running the SMT-LIB benchmarks. Z3 accepts input as a file whereas Yices only will read from standard input. The tests were all run on elf.bu.edu which has a 32-core 2.7GHz Intel Xeon E5-2680 CPU with 256GB of DDR3-1333 RAM and an nVidia C2075 Tesla GPU, running 64-bit Scientific Linux 6.

We measure the amount of time it takes a solver to decide a benchmark as the difference of time between calling the executable and its termination. Before logging the time, we of course check that it gave either "sat" or "unsat". Since elf is a shared computing node, it's plausible that each test will be arbitrarily preempted and its total runtime longer than how long it took to solve the benchmark. To address this, we only record the user time, or only time where the process was given access to the CPU. These solvers are CPU bound, so any time spent in kernel space will be somewhat irrelavent to decision such as outputting models.

2.0.1 Determining Capability

At the beginning of the project, we were unsure of how long we should expect a solver to finish a test. Some examples are proven on the order of milliseconds whereas some take hours. This made running the solver over the entire suite unfeasible for even moderate timeouts. In order to avoid the land mines that take ten minutes or more for both solvers, we decided to run each solver on a random sample of each logic.

Choosing a sample allowed us to quickly see where solvers perform best with high probability. Each benchmark in the random sample is distinct and all benchmarks have equal probability of being included. We used reservoir sampling as our selection algorithm. The sample size was originally set to about 200 benchmarks, but just to be diligent we increased the sample in the larger benchmarks (QF_LIA and QF_BV) to about half the population size. Fortunately, the patterns we observed were largely unchanged.

We first run both solvers with our lowest timeout over each benchmark in the sample. If one solver decides a benchmark within the timeout and the other doesn't we designate it the winner, otherwise we note both as finishing it. For any benchmarks that neither solver proved, we repeat this process for the next timeout for the unsolved benchmarks. If neither solver can prove a benchmark in any of the timeouts, we mark it as unknown.

2.0.2 Performance

The capability experiments were designed as simply an indicator of what logics each SMT could handle. Naturally, such an experiment based on efficiency will also give insight into how quickly each solver works for each logic, but we don't think it gives the whole picture.

For example, if both solvers can solve most of a logic within a second, should we still care whether or not one is faster? We think so, since a solver performing on the order of milliseconds would be a noticeable speedup in practice over one that works just under a second. By saying "in practice", we refer to users of tools that utilize SMT solvers as proving backends. If there are hundreds of constraints to prove, then small performance discrepencies become noticable to the user.

For these experiments, we run both solvers over the entire set of benchmarks with a 30 second timeout. Our intuition is that the capability tests give us a good idea of how quickly things can be decided past a second so 30 seconds will definitely find the better solver for a logic if there is one.

3 Z3

If the SMT Competition is any indicator of a where a solver sits with respect to the rest of the community, then Z3 is the current jack of all trades. It is used extensively as a backend in Microsoft's line of verification systems from verifying hypervisors with VCC to verifying C# applications using Boogie.

3.1 Logics and Features

Z3 decides a subset of many-sorted first order logic. Supported theories including linear and real arithmetic, limited non-linear arithmetic, arrays, bitvectors, quantifiers, and uninterpreted functions. User defined sorts may mix sorts from these theories and recursive datatypes are supported as well. The set of formulas that it accepts as input are first order formulas where each term's sort $s \in S$ where S is the set of all sorts across these theories.

Z3 can also be used as a MAX SAT solver, though this functionality is not directly available. Using parts of the C API, we can determine the maximum set of satisfiable sub-formulas of an expression. Unsatisfiable cores are supported as well, but the output is not guaranteed to be the minimal set of sub formulas that make an expression unsatisfiable[5].

Z3 has an advanced solving mechanism that can build solutions to satisfiable formulas given to it. This is known as model generation and is one of the more popular features of Z3.

3.2 Limitations

Despite being one of the best SMT solvers currently available, there remains some work ahead for Z3. The theory supporting non-linear arithmetic, for example, is cannot be used in combination with other theories such as bit vectors or arrays. The theory of non-linear arithmetic is also constraint to polynomials so formulas such as 2^n for non constant n are beyond the reach of Z3.

4 Yices

Yices supports the same logics as Z3, except for those involving non-linear arithmetic. It has rich support for determining MAX SAT, and includes unsatisfiable cores as well. Two significant features that Yices has is subtypes and dependent types. Subtypes allow you to define sorts such as S such that $S \subseteq T$, and T is the domain of a sort. Dependent types allow sorts to depend on other sorts in the context. Yices refers to sorts as types, so we use both interchangably when talking about Yices. Honestly, there are probably more implications to adding a dependent type system around first order logic than we can currently understand, so for this project we decided

To complicate things, SMT-LIB has no notion of subtypes nor dependent types. Instead of trying to modify the standard, we would like to focus on finding areas where Yices excel, and then make it accessible to the wider community by integrating it into an open format that can handle all the new SMT benchmarks available.

Yices also supports model generation to find solutions to satisfiable formulas.

4.1 Why Not Yices 1

Figure 2 demonstrates some of the problems that occur with Yices 1.0. The biggest disparity we notice is that Z3 solves almost 60% of our quantifier free linear arithmetic within a period of time that Yices cannot. This is interesting, as Z3 and Yices share the same fundamental approach to solving these formulas. Yices does have a small advantage with real arithmetic and difference logic, but overall it fairs unfavorably to Yices.

In Figure 3 we give some performance plots for comparing Yices with Z3. In these graphs, we plot the ideal function where Z3 and Yices are equivalent; the time taken to solve a benchmark is the same for both. Next, for each benchmark we create a point on the graph with an x value given by Z3s decision time and a y value given by Yices' decision time. The time units are in seconds and are plotted on a log scale. We will use this convention when comparing Yices 2 and Z3 in th next section. We include AUFLIA because it is a good stress test for theory combination since it encompasses arrays, uninterpreted functions, quantifers, and linear arithmetic. Of course, this is a difficult theory, but our results show that perhaps it's best to leave the advanced features to Z3.

Based on these results we aren't left with a compelling reason to integrate Yices with Z3. The authors at SRI recognized the need for revamping Yices and in 2008 they started building a prototype from the ground up in C. Among the reasons for the redesign were frustration with the complicated type system in Yices 1 along with the constant changes that happen in the C++ programming language[3]. This prototype is fairly undocumented, and we thought it would be an exciting experiment to see if it could hold a candle to Z3.

4.2 Yices 2 Features

Yices 2.0 is a lightweight SMT prototype that can only be used for satisfiability and model generation. It lacks any support for quantifiers, and outright rejects formulas involving nonlinear arithmetic. Its programming API was completely redesigned from the Yices 1.0 and offers no support for SMT-LIB 2.0. The class of formulas Yices 2.0 may decide is strictly a subset of Z3's input. Therefore, the best we can hope for in our comparison is that Yices 2.0 can solve some formulas faster than Z3.

In the talk at SRI where he outlined the reasons for making Yices 2.0, Bruno Dutertre lists domains interested in using Yices primarily as a constraint solver, not for proving theorems. If this influences development, Yices 2 may be viewed as a more domain specific SMT solver in constrast to the feature rich Z3. In the comparison section, we outline some constraints from the ATS Programming Language that we found to be better suited to Yices 2.

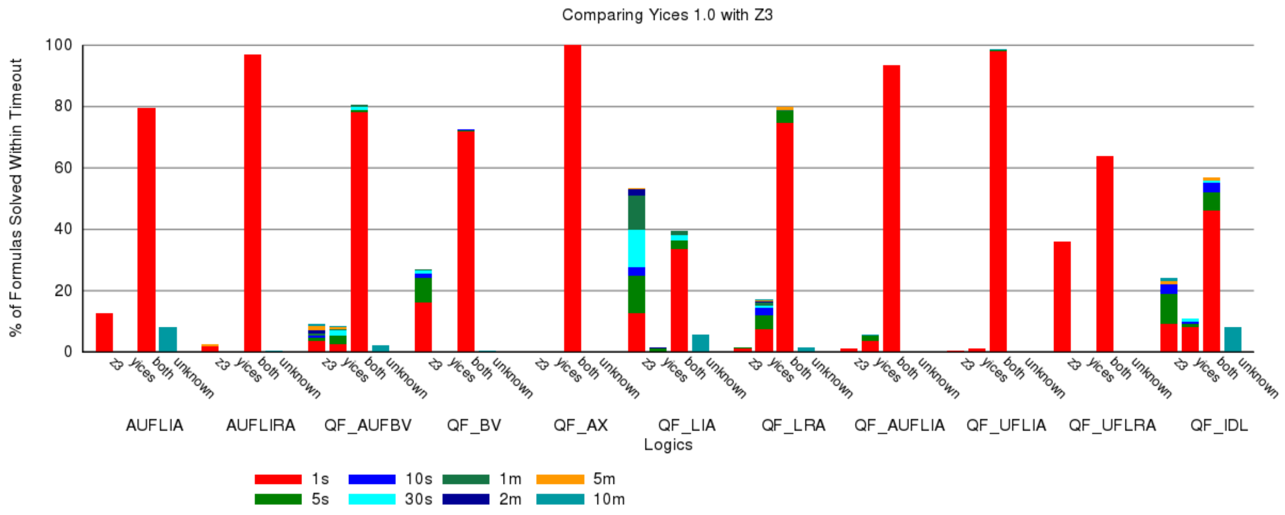
5 Comparison Results

Using the experiments described earlier, we evaluated Yices 2 and Z3 on the set of logics that both can decide. That is, all of the quantifier free arithmetic, bitvector, and uninterpreted functions benchmarks. The QF_UFBV couldn't be ran since Yices lacks support for some of the functions. We noticed that Yices 2's API does have support for these now, so they could be added into our solver.

5.1 Capabilities

Yices 2 demonstrates some significant advantages over Z3 in a number of logics. Figure 4 shows it edging out Z3 in almost every logic except for some involving uninterpreted functions, most notably linear difference logic with uninterpreted functions. This comparison is nice, but the vast majority of functions can be solved within a second. This sets motivation to see whether one solver consistently performs better than the other within this 1 second timeframe.

Figure 2: Yices 1 Consistently lags behind Z3



5.2 Performance Breakdown

We were surprised to see Yices 2 has an advantage over Z3 for some logics. Figures 5 and 6 give several plots of the kind we used for testing Yices 1 vs Z3. A large concentration of points beneath the "equivalent" line shows formulas that Yices can solve faster. Some logics are not so cut and dry, however, as quantifier free bitvectors and quantifier free linear arithmetic still have plenty of benchmarks that are decided by Z3 than Yices. Nonetheless, we claim that we can improve on this by using Yices as an "optimzer" for Z3. The intuition goes that if we ran each SMT solver concurrently, the ideal result would be that the time to solve each benchmark would be the minimum of their x,y values on our performance plots.

5.3 Case Study: Linear Inequalities in ATS

As a brief aside, in a related project we evaluated the constraint solver in the Applied Type System programming language with Z3 and Yices. We found that Yices was faster at solving these constraints. All constraints we place on runtime variables in ATS are translated into conjunctions and disjunctions of linear inequalities of the following form.

$$a_0 + a_1x_1 + a_2x_2 + ... + a_nx_n \bowtie 0$$

If a program is well typed, then we say that all of its constraints are valid; they are satisfiable under every model. We can prove this by taking the negation of the constraint and passing it to an SMT solver. If it returns SAT, it has found a counter-example to our claim. If it is UNSAT, then this proves our implementation is valid under this constraint. We included a small SMT2 script in our test directory to try it out with our composite solver.

6 Composite Solver

We already mentioned the possibility of optimizing Z3 by using Yices 2. This is the approach we took to build our composite solver which runs both Z3 and Yices in parallel. Since SMT-LIB 1.2 is deprecated, we decided we would want Yices 2 to be more accessible to the greater SMT community by integrating it into an SMT-LIB 2 parser developed by Alberto Griggio [1], a member of the MathSat team.

6.1 Case for SMT Lib 2

SMT Lib 2 is a worthwhile improvement because it enables a more interactive approach to solving. In the old language, benchmarks were declared by a single formula and then checked for satisfiability. This makes granular reasoning difficult. It is not very helpful when your tool discovers a counter example, but it doesn't help you in anyway to solve the problem.

Indeed, both Z3 and Yices support more modular operations like (push/pop) and incremental assertions in their internal APIs. From a tool standpoint, this is favorable as you can add constraints one at a time and check for satisfiability at each step. Thus, you spot problematic formulas faster and are able to give better details to the user. Since SMT solvers are so low level, the user may be some verification tool. Nonetheless, more precision

[1] `https://es.fbk.eu/people/griggio/misc/smtlib2parser.html`

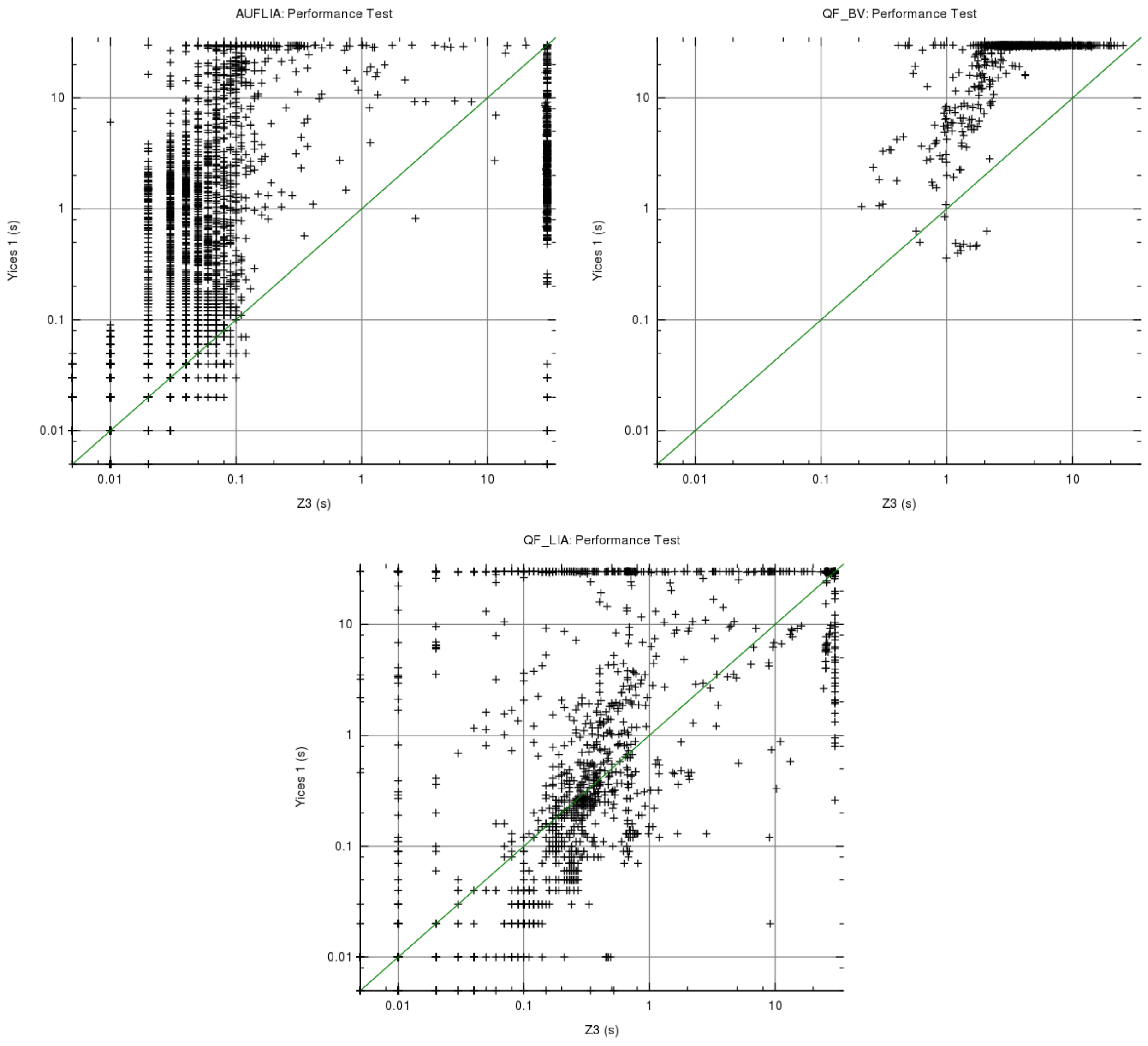
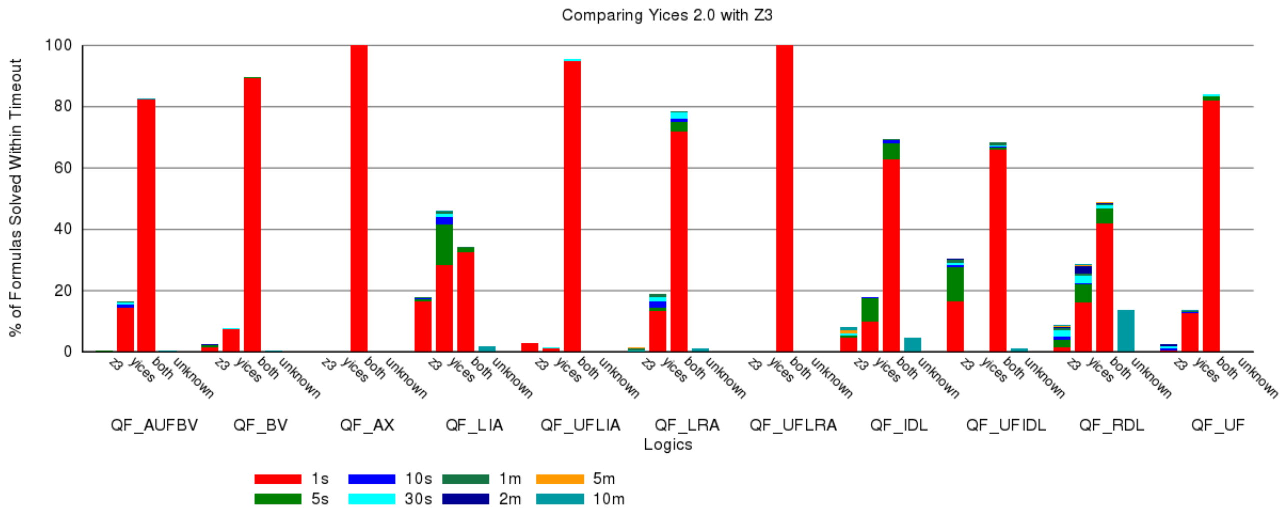


Figure 3: Yices 1 Struggles Against Z3

Figure 4: Yices 2 Shows Promise



in what makes a context of assertions satisfiable or not is useful for any system implementor.

Supporting SMT LIB 2 gives an interesting challenge for us for a composite solver. SMT2 files contain potentially many calls to (check-sat) and our goal is to produce the solution to these calls using two solvers. We could prepend each answer with the solver's name, but our goal is to make a single solver as easy to use as either Z3 or Yices. Therefore, a controlling thread receives answers from worker threads who communicate read from the solvers' output individually. Ever solver's answer has an increasing index that the controller uses to see whether or not the answer is for the current index. This allows the controller to disregard a slow solver's output and prevent duplicates being displayed to the user.

The program terminates when one worker thread sends a "finished" message indicating its solver has terminated. If the solver just outright fails, our simple prototype won't recognize it. Unfortunately, our composite solver blocks the output of model generation functions, but this could be resolved somewhat easily.

6.2 User Manual

The following is a brief description of how to compile and run our solver.

6.2.1 Compiling

We use a couple of libraries that are included with our source code. It has been tested to compile and run on csa2.bu.edu, but it will probably work on any 64-bit Linux setup. A README file in our project's src directory goes over compiling. We require that you have Z3 installed. It turns out z3 is installed on csa2.bu.edu.

6.2.2 Test Cases

Our test cases are rather small, but demonstrate that the solver works. The first couple of tests are simple cases with one or two checks. There are a couple of tests called bounds and tally that are constraints taken from ATS programs. Bounds is just a contrived example of the same constraint repeated over and over, whereas tally verifies some arithmetic identities. The final test case is from the SMT-COMP 2012, but our solver does not finish (the finish message is not sent or received for some reason). You can see the interaction between the controller and slow solvers, but after the last assertion is checked our master thread hangs.

7 Discussion

In this project we implemented a basic parallel composite solver that sought to speed up some formulas users may encounter with Z3 using the Yices 2 prototype. For SMT2 files, we think the approach of just running two solvers and waiting for one to eventually finish doesn't take advantage of the opportunity to really parallelize solving.

Consider that the semantics of the SMT2 language are closely tied to the functional language Lisp. This means that a defined term should be immutable. Indeed, if I try to redefine a function in a smaller scope, a solver like Z3 will throw an error. Since push and pop are used extensively in SMT-COMP, we can view the whole benchmark as a set of independent jobs that need to be solved. The

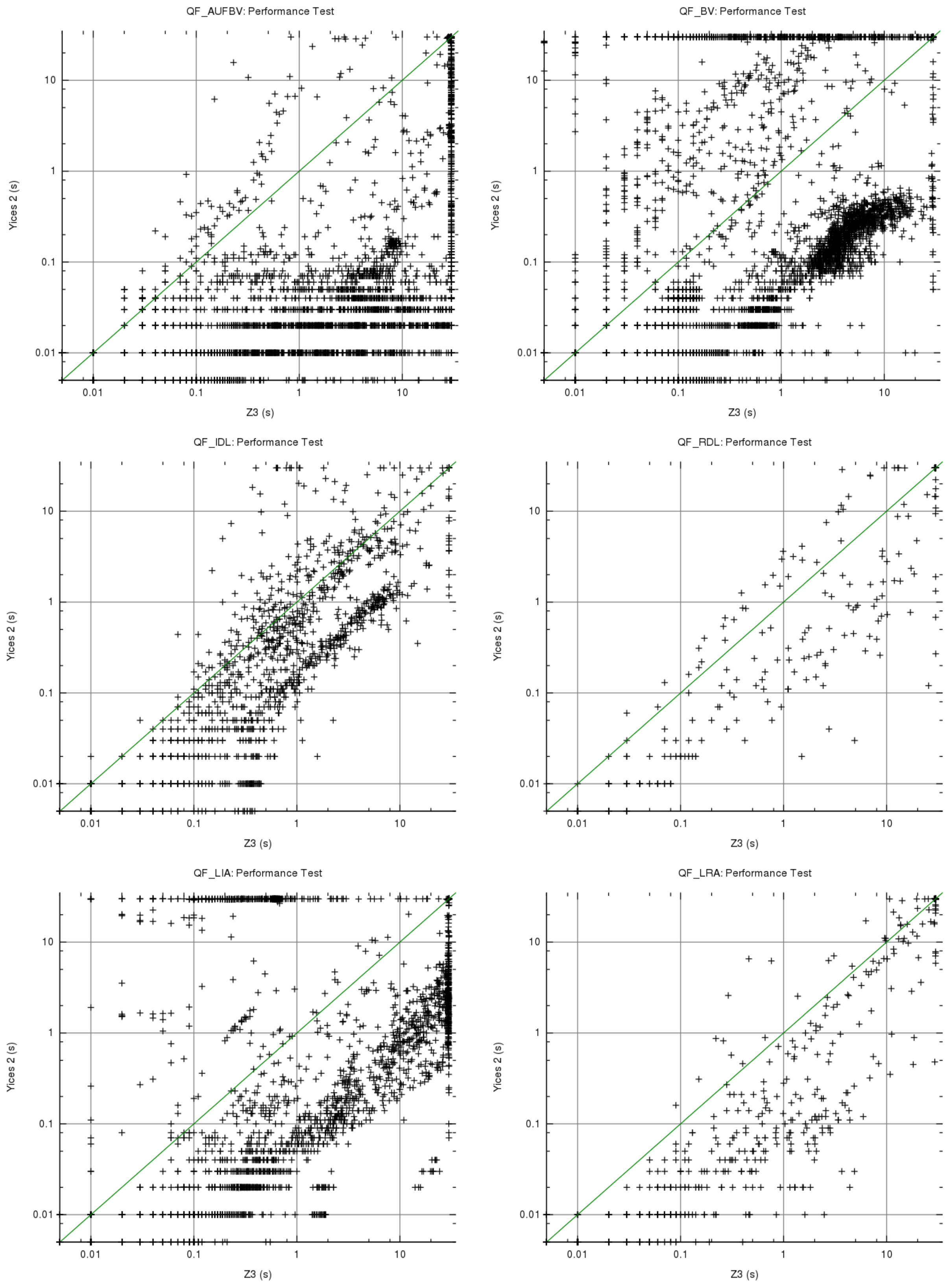
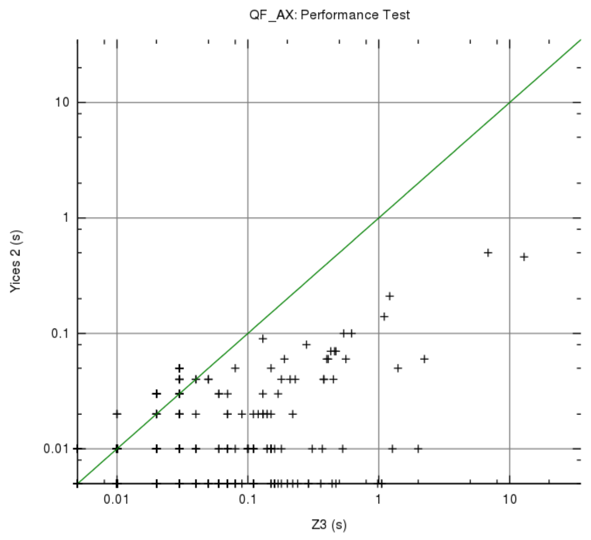


Figure 5: Where Yices 2 Excels over Z3



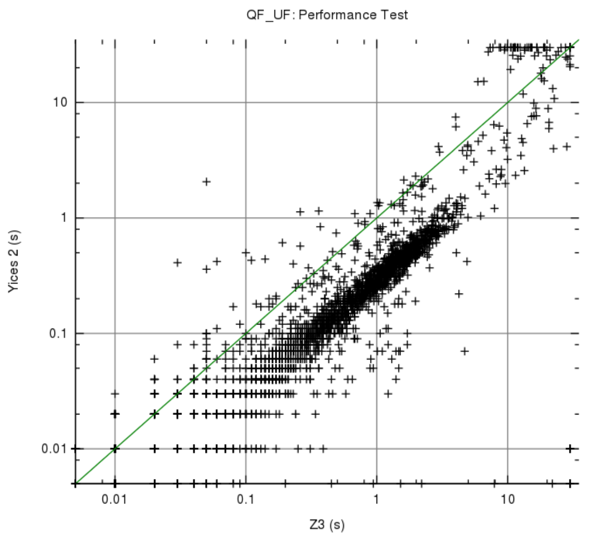


Figure 6: More examples of speed ups with Yices 2

independence is useful because it could allow us to parallelize lots of decisions.

We sketch an idea of how to do this. Given the use of push and pop to add and retract assertions, each push represents a new solving job we could queue along with the current context. The worker could then go to the corresponding pop (branching for each push/pop along the way) and resume solving. As long as the order in which check-sat commands are called is preserved in the output to the user, this could possibly speed up the decision procedure.

References

[1] David R Cok, Alberto Griggio, and Roberto Bruttomesso. The 2012 smt competition. 2012.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[3] Bruno Dutertre. Challenging problems for yices. Seminar Slides, 2011.

[4] Bruno Dutertre and Leonardo De Moura. Integrating simplex with dpll (t). *CSL, SRI INTERNATIONAL, Tech. Rep*, 2006.

[5] Leonardo De Muora. Getting a ”good” unsat-core with z3. `http://bit.ly/Y4pkqz`. Accessed: 2013-04-15.

[6] Microsoft Research. Z3 rise4fun tutorial. `http://bit.ly/13dBcqf`. Accessed: 2013-02-27.